The code document's structure and analysis

Stuart Reeves
School of Computer Science & IT
University of Nottingham, Jubilee Campus, Wollaton Road,
Nottingham, NG8 1BB
str@cs.nott.ac.uk

April 21, 2006

Introduction

The purpose of this paper is twofold. Firstly it presents a preliminary and ethnomethodologically-informed analysis of the way in which the growing structure of a particular program's code was ongoingly derived from its earliest stages. This was motivated by an interest in how the detailed structure of completed program 'emerged from nothing' as a product of the concrete practices of the programmer within the framework afforded by the language. The analysis is broken down into three sections that discuss: the beginnings of the program's structure; the incremental development of structure; and finally the code productions that constitute the structure and the importance of the programmer's stock of knowledge. The discussion attempts to understand and describe the emerging structure of code rather than focus on generating 'requirements' for supporting the production of that structure. Due to time and space constraints, however, only a relatively cursory examination of these features was possible. Secondly the paper presents some thoughts on the difficulties associated with the analytic—in particular ethnographic—study of code, drawing on general problems as well as issues arising from the difficulties and failings encountered as part of the analysis presented in the first section.

The following section discusses briefly the background to the program's production, which has relevance for the reader's understanding of the rationale behind the structural decisions that were made.

The program and sittings

I came across *diffusion limited aggregation* (or DLA, and also known as a Brownian tree) in a popular science book. The book depicted the output from a DLA program and gave a brief description of the process involved in generating the image (as in Figure 1, which is actually the output from my program). Diffusion limited aggregation is a process whereby 'particles'—pixels in this

case—are stepped through a random walk, 'sticking' to already existing particles within a plane. The process is seeded with a pre-existing particle placed in the centre of the plane from which the two-dimensional, plane-bound image is generated.

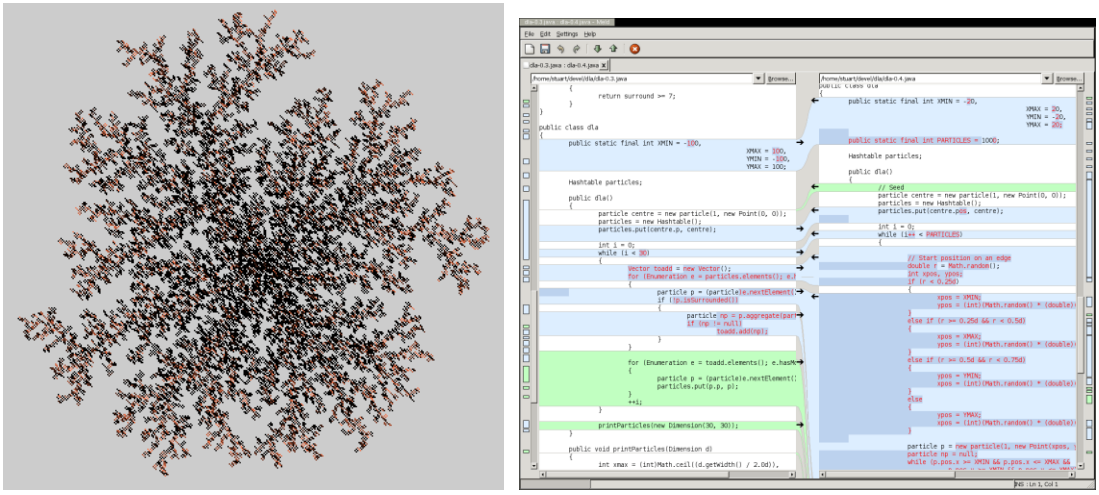

Figure 1: Diffusion limited aggregation (left), `meld` screenshot (right)

The program was written 'from scratch' in Java over the course of couple of weeks or so. Various resources were drawn on during this work, such as conversations, 'workings' in the form of sketches and diagrams (as shown in Figure 2), the Java API documentation, other implementations of DLA available on the internet, and so on. A series of versions (this term is used sparingly for reasons discussed shortly) was kept from the initial attempt to the final working program (`0.1` through to `1.2`) along with occasional compiler output and notes that seemed particularly pertinent at the time. In order to examine the way in which the code structure developed, and instead of examining program development in real-time (e.g., by video or screen capture data), the differences between these 'versions' were studied as a way to perform some introspective analytical work on a data set that was already known intimately. This analysis was done using the program `meld`, which is a visual diff-ing tool that provides a graphical representation of differences between two (or three) files (see Figure 1 for a screenshot).

This 'version numbering'—e.g., the decision and practical work of moving from version `0.1` to `0.2`—was iterated according to a particular emerging strategy of capture termed here as 'sittings.' A sitting came to be constituted of at least a temporally continuous session of work at the keyboard, or at most a small number of these sessions with minor breaks in between. It is of note at this point, however, that the notion of a sitting is however distinct from the term 'version,' as it is used in common software development parlance. A version of a program's code in this sense may be the result of many, many sittings and possibly a long development process (see Figure 4 for a graphical illustration of this point).

The reasons for the sitting's lifetime being as it was varied. A hard prob-

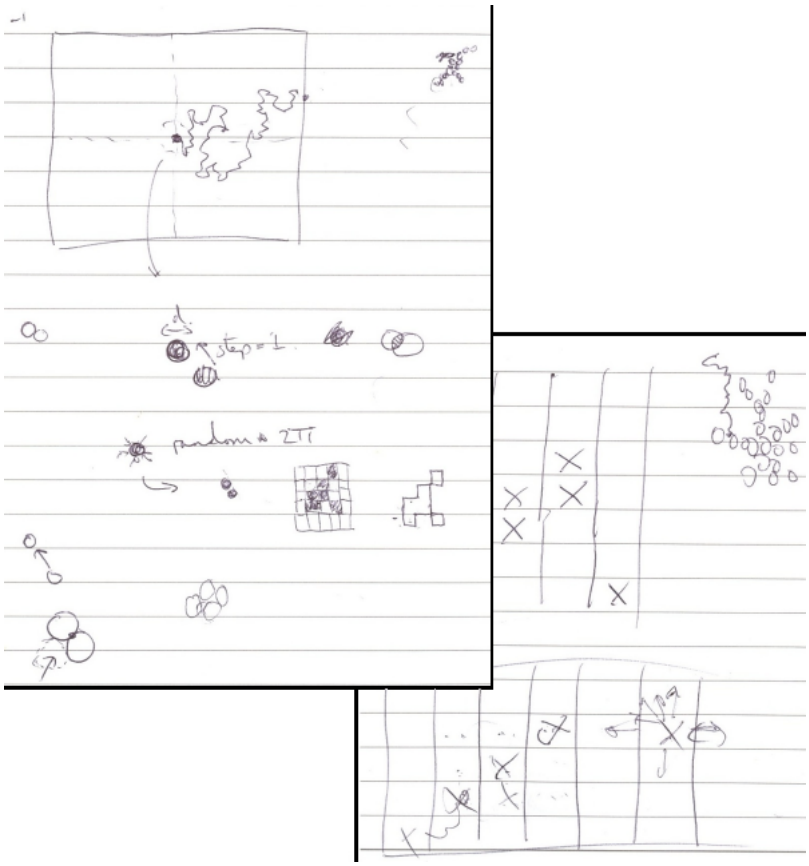

Figure 2: Two pages of sketches and drawings

lem may have been encountered ('getting stuck'), bringing about the end of the sitting, or perhaps the sitting naturally came to a conclusion as part of successfully 'adding a feature.' Equally, the program's state at the end of the sitting could be varied, such as it being anything from compilable and runnable to not-compilable and not-runnable (i.e., crashing). In spite of these varied end states, a prime feature of a sitting's edits seemed to be that some 'conclusion' was reached in terms of the task-at-hand as well as the visual state of the code. For example, syntactically incorrect or incomplete statements ('hanging lines' of code) were *not* left as such. As both a byproduct of this and a further feature of the maintenance of conclusive sittings, by the end of a sitting the visual (whitespace) structure of the code was always 'correctly' and consistently indented, spaced and the like. From the general features detailed here, then, is clear that creating some discernible, legible 'tying up of ends' or leaving the code in an otherwise visually 'tidy' state seemed to be important in the conclusion of a sitting.

Beginnings

The notion of the sitting is intimately linked with the programmer's methods of 'keeping track' and using placeholders as part of a concerted effort to maintain the trajectory of the code as it is worked upon. It is these issues that shall be examined here in more detail, but it is also worth highlighting at this point that the analysis delves into some low-level technical detail regarding the issues this paper attempts to draw out, however this itself is an interesting point that will be addressed in a later section.

The first 'sitting' resulted in the code that follows in Listing 1 (sitting `0.1`). The entire listing of this sitting has been reproduced as it contains some interesting features that shall be examined. The code contains two classes: `particle` and `dla`. The `particle` class is notionally correspondent to the 'particles' that are randomly walked through the space. It contains a constructor that accepts two arguments, whereas the `dla` class contains a constructor and the `main` method (i.e., the entry point for the program). It is of note that this code was not compiled at all during the sitting, and compilation was only performed at the start of the next sitting some time later (and generating multiple compilation errors).

The functionality the program attempts to express is as follows. When the entry point `main` is processed, a new `dla` object is created. This `dla` object creates a new `particle` object with (unused) `diameter` and `step` arguments set to `float` values of `1.0` and `0.0`. The `particle` constructor is called during this creation, picking a random number (`r`) and filtering this with conditional statements through to four possible outcomes, the purpose of which is to choose a random side of a cube to as part of the `particle`'s start position. Each outcome (e.g., `if (r <= 0.25d && r < 0.5d)`) then assigns to the member fields `xpos` and `ypos` a value that corresponds with a random placing along the chosen side. This is illustrated in Figure 3. After this selection, the `particle` (`centre`) has its fields `xpos` and `ypos` directly set to zero. Finally, the `Vector` object `particles` is created and the `dla` constructor exits, leading to the exit point for the whole program.

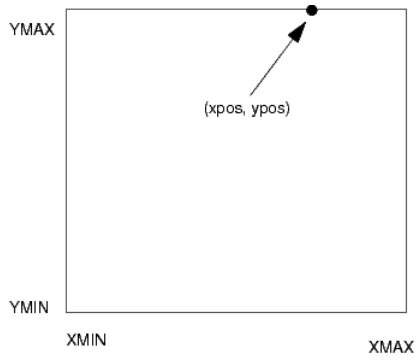

Figure 3: `particle` start points

Listing 1: Sitting 0.1

```
import java.lang.*;
```

```
class particle
{
      public double xpos, ypos,
             xdir, ydir;
      double diameter, step;

      public particle(double diameter, double step)
      {
            double r = Math.random();
            if (r < 0.25d)
            {
                  xpos = XMIN;
                  ypos = Math.random() * (YMAX - YMIN);
            }
            else if (r >= 0.25d && r < 0.5d)
            {
                  xpos = XMAX;
                  ypos = Math.random() * (YMAX - YMIN);
            }
            else if (r >= 0.5d && r < 0.75d)
            {
                  ypos = YMIN;
                  xpos = Math.random() * (XMAX - XMIN);
            }
            else
            {
                  ypos = YMAX;
                  xpos = Math.random() * (XMAX - XMIN);
            }
            this.diameter = diameter;
            this.step = step;
      }
}

public class dla
{
      public static final int XMIN = -100,
                              XMAX = 100,
                              YMIN = -100,
                              YMAX = 100;
      particle centre;

      Vector particles;

      public dla()
      {
            centre = new particle(1.0f, 0.0f);
            centre.xpos = 0;
            centre.ypos = 0;
            particles = new Vector();
      }

      public static void main(String[] args)
      {
            dla d = new dla();
      }
}
```

Although sitting `0.1`'s code here contains only 59 lines, the skeleton structure of the `dla` and `particle` classes are not radically modified by the final sitting and indeed still persist in `1.2`'s 838 lines of code. This initial sitting 'does nothing' (as shall be discussed shortly) and yet functions as a constructional 'sketch' or trajectory for subsequent sittings, as well as in some sense being the product of the programmer 'thinking out-loud.' It provides a structural orientation for the programmer in subsequent sittings as the program is developed. This orientation is made available to the programmer by self-accountable and self-legible features of the way in which the programmer constructs this initial sitting's work, particularly in the use of whitespace, com-

ments, formatting and other elements of 'good practice.' Thus the programmer is assisted in rapidly being able to reacquaint him/herself with some thing previously written at the next sitting. That the code does not compile is a further indication of this sitting's work being 'sketchy.'

As mentioned, major parts of the code at this stage appear to 'do nothing' and perhaps could be seen as vacuous and insubstantial. This is not necessarily the case, however, as the code here exhibits a stage of development where declaration and structure are employed to prospectively 'scaffold' the next sitting by furnishing that sitting with 'placeholders' for further action. Besides the non-compilable, non-runnable state of the program, this is exhibited in logically 'useless' material, such as the `particles Vector` or some of the member fields such as `particle centre`, the `doubles diameter` and `step`, and so forth. These declarations serve no purpose here-and-now, however they each have a prospective dimension for the programmer's work in that they help orient the next sitting in which the concepts embodied by their declarations may be logically 'filled out.' It is obvious that the naming of these declarations is similarly vital for their legibility[1] as well as their position within the structure of the code (e.g., that `particles` is declared within the `dla` class rather than within the `particle` class). Given that the future sitting may be temporally distant, certain particulars and intentions developed here-and-now within this sitting may also become fuzzy and indistinct at that later date, and so the programmer, in their 'tying up of ends' at the end of the sitting, attempts to provide enough scaffold—cultivated in these 'useless' declarations—to ensure that during the later sitting the code's trajectory is as legible as possible.

As an example we can trace through the changes made to the 'useless' `Vector` named `particles`. As written in sitting `0.1`, it forms a placeholder for some kind of container for `particle` objects. In sitting `0.2` this container's type is changed from being a `Vector` to a `Hashtable` of `particle` objects instead. Also during this sitting the constructor of the `dla` class is functionally filled out, in which (as shown in Listing 2) `particle` objects are `put()` into `particles` as part of a nested `for` loop (not shown). Thus `0.1`'s declaration of `particles` begins to serve a concrete logical purpose within the context of the rest of the code.

Listing 2: The dla constructor, sitting 0.2

```
Hashtable particles;

public dla()
{
      particle centre = new particle(1, new Point(0, 0));
      particles = new Hashtable();
      particles.put(centre.p, centre);

      while (true)
      {
            // …
            particles.put(p.p, p);
      }
}
```

Button and Sharrock's account of programming work has highlighted the "screen practices" of programmers (that is, practices produced from screen-based work). In particular they pointed out that "writing code does not consist

[1] See [2] for a relevant discussion on temporary naming of variables.

of writing the complete code straight off" and furthermore noted how the programmer may write out the "simplest possible case first" as part of keeping track and not visually—and thus 'mentally'—"losing sight" of the code's unfolding development [2]. These screen practices are similar to the sketch-like, "simplest case first" work practice as exhibited here in the first sitting. Rather than write the whole detailed structure out in one pass, the programmer employs the prospective qualities of 'useless' declarations in order to avoid losing sight. These prospective declarations provide legibility not just for the immediate here-and-now unfolding of the code *but also* for future sittings and the trajectory of the structure of the entire program. In other words they are self-orienting ('keeping track') practices that exhibit and embody for the programmer the course of the unfolding of code.

Finally, something that is also of note for this first sitting is the nature and role of the commonsense notion of 'plans' in writing code. Here Suchman's observation on the conflict between plans and the actual, lived work of actions carried out ostensibly in support of such plans [7] is of relevance. Whilst large code projects do indeed feature detailed plans in the form of specifications, sketches, more formal diagrams (using, say, UML) and design documents (to name a few), these plans cannot encapsulate the contingencies the programmer continually encounters. Plans can instead be seen as resources for action, with the coupling between plans and actions having varying degrees of homogeneity. For example, unlike some forms of activity in which the conflict between plans and situated actions is more extreme and pronounced (e.g., Suchman's example of the kayaker), the programmer is able to use 'sketch' code and 'thinking out-loud' code (as exhibited here) to situatedly establish elements of a 'plan' as part of the concrete lived work involved in programming. The code thus can be part of the construction of a plan as well as the activity involved in the manifestation of that plan. The development of the sittings in this study foreground the use of plan-like sketch code that establishes and maintains a trajectory, i.e., a plan of sorts, however it is of note that within this particular study there are no explicit commonsense planning features such as specifications or design documents beyond a couple of hand-drawn sketches (mostly owing to the simplicity of the task).

Incrementing

Code is often ever-developing and sometimes seemingly never finished[2]. Although structure is fundamentally addressed by the work of the programmer, and at times is produced 'en masse' (e.g., as in the creation of sitting `0.1`, where the entire initial structure was produced), incrementally adding features is also a staple everyday practice of programming and creating structure. Incremental code productions accrue across multiple sittings, slowly modifying the program and enabling the code document's progression along its trajectory in a closely guided way. Placeholders and otherwise 'useless' declarations are prospectively employed over the course of this incremental activity.

In writing a program, the programmer often exploits the modularisation and encapsulation facilities provided by the language's constructs (such as Java's object orientation with features like classes, inheritance, polymorphism,

[2] See many Free Software projects for this effect.

etc.). In spite of the abstraction and separation afforded by such constructs, the actual everyday work of writing code is a highly contingent activity in terms of the way in which that modular structure is produced incrementally. For example, in order to add an argument to a method, code calling the method must be brought up-to-date with this change; this edit cannot be saved for a 'later date' if the programmer wishes to test run the program at the current time. Thus, although the logical structure may be modular and abstracted, the work that goes to make that modularity is most definitely not.

Firstly, and following on from the previous section, we shall examine a simple example that illustrates the way in which the placeholder quality of certain declarations is exploited during incremental change. Sitting `0.2` introduces a member field `Point p` into the `particle` class (see Listing 3). This field was functionally conceived for storing the Cartesian point the `particle` object was located at, meaning that at first it would be set with a chosen starting point (i.e., on an 'edge,' see Figure 3), and then used to store the updated position according to the random walk. At this sitting the variable `p` again 'does nothing'; it is merely set in the constructor for the `particle` class. It is only by sitting `0.4` that this variable is referenced anywhere else in the class (in the `step()` method), and, notably during this sitting is relabelled as `pos`. A reason for this is the visual 'collision' between another `particle` variable, `pp`, that is used in the same `step()` method that `p` comes to be employed in. The placeholder feature of `p`, then, is used to denote temporarily some future member of data and piece of functionality (i.e., the `particle`'s current position) that is only employed within the `particle`'s functional workings at a later sitting. For the time leading up to this sitting it also does a 'job of work' for the programmer in the method that calls the `particle` constructor. `particle` objects are created in the `dla` class, as seen in Listing 2, enabling the programmer to structure `particle`-creation code in that class *as though it were a feature of the* `particle` *class*, regardless of that part of the code 'doing nothing.'

Listing 3: Placeholder field, sitting 0.2

```
Point p;
public particle(int diameter, Point p)
{
      // …
      this.p = p;
}
public particle step(Hashtable particles)
{
      // …
      particle pp = null;
      // …
}
```

The previous example examined only a few modifications between sittings `0.2` and `0.4`. Listing 4 illustrates another example of incremental development, albeit traced over the course of *all* the sittings[3]. Here the `while` loop within the `dla` class constructor begins as a endless loop (`0.2`). At this stage, the content of the loop iterates through the `Hashtable` of existing `particle` objects, calling the `particle`'s `aggregate()` method on each and adding

[3]The following description of this development may be difficult to follow, and so the sittings have been made available at `http://www.mrl.nott.ac.uk/~str/pages/dla-applet/dla.html`

new particles to the Hashtable. By sitting 0.3 the loop is bounded by a counter and literal value, and the loop's contents are identical to 0.2. Subsequently in 0.4 this loop becomes bounded by a 'hard-coded' constant and its contents are modified somewhat. Here in 0.4 a new particle object is being created at the start of each loop, which is itself declared with the edge selection procedure (as in Figure 3). This particular block of code has been moved inside the while loop, and a further inner while loop has been constructed. This inner while calls the step() method (i.e., to 'step' the particle through its random walk) repeatedly on the particle created earlier in the loop until conditions determine that it has either been aggregated (i.e., 'stuck' to another already-existing particle), or 'walked' outside the bounds of the square's edges. Sitting 0.5 has the (now outer) while loop bounded by an argument (iterations), which, some sittings later is assimilated into a new aggregate() method[4]. By sitting 1.0, the loop's contents have been expanded and has been transformed into a for loop. Given this transformation, there are, however, the vestiges of the edge selection and the inner loop that calls step() on the particle objects repeatedly.

It is important to note that for all of the sittings after 0.2, the loop counter variable i is not referenced by any statements within the loop; that is, the variable's use is completely self-contained with respect to the code reproduced here.

Listing 4: Developing the loop

```
0.2:
public dla()
{
      // …
      while (true)
      {
            // …
      }
}

0.3:
public dla()
{
      // …
      int i = 0;
      while (i < 30)
      {
            // …
            ++i;
      }
}

0.4:
public static final int PARTICLES = 1000;

public dla()
{
      // …

      int i = 0;
      while (i++ < PARTICLES)
      {
            // …
      }
```

[4]This naming is potentially confusing. Initially the particle class had an aggregate() method, but this was later changed to be labelled step(), whereas here the dla class has a (separate) method named aggregate(). The change comes about due to the change in functionality of each class, however there is not enough space to deal with this development in depth.

```
}

0.5:
public dla(int iterations, ...)
{
      // …

      int i = 0;
      while (i++ < iterations)
      {
            // …
      }
}

...

1.0:
public dla()
{
      // …
}

public void aggregate(int iterations, ...)
{
      // …
      for (int i = 0; i < iterations; ++i)
      {
            // …
      }
}
```

One of the most obvious aspects of this series of code fragments is the way in which the loop develops incrementally across sittings `0.2` to `0.5`. There is an apparent 'slowness' in this incremental development given the very simple segment of code that is actually being modified. There is no technical reason why its final form (i.e., `1.0`'s generalised and argument-bounded `for` loop) could not have been reached at the first or second sitting. Besides the `iterations` argument that is added, *logically* there are no contingencies developed upon the surrounding code, and yet the loop's development is 'slow.' The gradual changes made to the code in this sequence do not *just* derive from the (virtually non-existent) logical contingencies local to the sitting, but also from contingencies upon the way in which the program's structure unfolds *across sittings*. In other words, the surrounding code and the content of the loop's code has a changing state and functionality that is developed as part of the concerted work involved in the loop itself. The loop is not developed and does not have a trajectory that is in isolation from its local surroundings. This is evident when we examine the loop's placeholder qualities and its use in the ongoing development of the code.

This sense of the loop as 'placeholder' can be seen as part of the programmer's attempts in 'just getting the code to work.' The initial construction of the `while` loop has the condition `true`, meaning that it will never exit the loop, and given that there are no `break` or `return` statements, it could be seen as a highly unwieldy formulation for the context of the code that surrounds it. However, it is 'just enough,' adequate and fit for purpose within this sitting; i.e., it simply does not matter at this sitting that the loop is unbounded, and has a purely utilitarian adequacy for the task at hand within the sitting it was produced in. But at the same time the loop also has a status as a placeholder or scaffold for subsequent action as the loop develops in later sittings. As the code is worked on in later sittings, bounding the loop becomes useful for lim-

iting output as print statements for debugging are introduced. Subsequently, the loop is altered to enable bounds specified as arguments, which is implemented as part of a series of other introductions of arguments into the code. As such, then, the development of the loop is continuously contingent upon the trajectory of the code as a whole.

Structure as a stock of knowledge

The structure of code is constituted from the constructs provided by the language the code is written in. A program may be structurally and functionally different from all others, and yet at a certain level is created from identical constructs to other programs. These constructs strictly define the basic building blocks of structure, and, typically due to pedagogical instruction, ability and practice, the programmer becomes skilled in expressing computational tasks in this language of constructs. A question here to explore is how repeated (or repetitive) instances of use of constructs across different contexts—which by their nature are highly specific—incrementally inform and build the programmer's generalised and adaptable stock of knowledge that is then consulted in later programs.

One of the major jobs of work for the novice programmer is becoming familiar with and competent at producing semantically and syntactically correct code statements. This is part of the novice's development of a mastery of 'good practice,' or appropriate 'ways to do things' in code. For example, in Java this would involve obtaining a handle on the essential patterns of structure that can created from basic language constructs like `for`, `while`, `if`, and their appropriate context of use, such as understanding when to use `switch` statements instead of `if (...) else if (...)` and vice versa. In addition to also cultivating a sense of whitespace and formatting 'style', the work might involve developing a familiarity with Application Program Interfaces (APIs) and essential tools of the trade (such as the Standard Template Library for C++).

Some elements of this stock of knowledge are exhibited within various sittings of the DLA code, such as in the use of `for` loops. A `for` loop is a simple control structure that enables bounded loops to be specified in a compact way, e.g., `for (int i = 0; i < 10; ++i) ...` Consider the following less typical `for` loop that appears in sitting `0.2`. In essence, the loop here enables the iteration through of all the 'elements' of the `Hashtable`, meaning that each value (i.e., `particle` object) of the hash's key-value pairs is processed:

Listing 5: Loop from sitting 0.2

```
for (Enumeration e = particles.elements(); e.hasMoreElements(); )
{
      particle p = (particle)e.nextElement();
      // Do stuff with p ...
}
```

There are many different ways to extract each value from `Hashtable`s and `Vector`s[5], however this particular, single technique is one that is repeatedly used over the course of the sittings. Immediately below the code of the `for`

[5] Any Java API classes that support the `elements()` method can be dealt with in this way, i.e., iterating through a returned `Enumeration` object.

loop that has just been discussed, for example, is another instance of this `for` loop 'pattern,' which in this case is used to process a `Vector` (named `toadd`):

Listing 6: Another loop from sitting 0.2

```
for (Enumeration e = toadd.elements(); e.hasMoreElements(); )
{
      particle p = (particle)e.nextElement();
      // Do stuff with p ...
}
```

By the final sitting these two instances of `for` loops have been replaced by other constructs, however in two other places within the final sitting[6] these kinds of 'stock' productions of loops have appeared yet again. Within the DLA program, these characteristic formulations of loops are a product of the programmer working with certain data structures like `Hashtable`s and `Vector`s. What has been established is a certain familiarity with 'way of doing things,' which in this case is a 'way of doing things with `for` loops and `Enumeration`s,' and is a doing that is repeatedly employed at different points in the ongoing development of the DLA program's code document. It is a tried and tested part of the programmer's stock of knowledge.

Repetivitity is a prime way for the programmer being able to introspectively notice their own patterns, as in the particular use of the `for` loop. This kind of repetitivity does not imply that programming becomes a drudge[7], but is rather an essential part of the construction of the stock of knowledge. This is comparable to many craft and craft-like activities.

As experienced programmers develop their skills further, their stock of knowledge can extend beyond including just certain small-scale patterns of language constructs to favoured large-scale 'design patterns.' These are in a crude sense more sophisticated 'ways to do things' in code. There is extensive professional recognition of the notion of these extended patterns of structure (at least within object-oriented software design) as evidenced by the popular and well-known account of this within object-oriented software design and construction [6].

Summary

Some of the features highlighted within this brief examination of a relatively small segment of the changing code document have been: the sketch-like quality of the initial sitting, and implications this has for the code's trajectory and structure; the code's trajectory as an incremental accomplishment that is maintained and tended to over the course of a series of sittings; and finally the role of the stock of knowledge in the programmer's incremental production of structure. The next section examines the analytical issues and problems this analysis has in particular foregrounded for the author.

---

[6] `ConfigReader.find()` and `dlac.paintComponent()`, as found in `1.2`.

[7] Turing presciently noted that

> [Programming] should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself. [9, page 18]

Analytical problems

The text denoted as 'code' is a steadily *changing* document of the programmer's daily work. Even given the detailed scaffolding that is part and parcel of everyday programming, code requires the vulgar competencies of the reader to make sense of its structure and meaning. The code document is not a 'synopsised' and peculiarly condensed record of the lived work of its production, but rather the document is instead a synopsis and condensation of the *intended purpose* of the code at the point it is examined. Typically the code's synoptic account of 'work' will not contain any information regarding the code's trajectory, unfolding structure and thus the concrete work in which it was produced. Furthermore, all code has a context of production, whether this is as a piece of hobbyist work or part of an industrial control program, a context which also will have left potentially few traces within the document itself. This section addresses the legion practical difficulties posed to the analyst when examining the code document and everyday activity of the programmer, with reference to the problems encountered during the analysis presented in the preceding sections.

Vulgar competence

It is often stated or implicitly assumed within ethnographic—particularly ethnomethodological—studies that "*the* challenge for the work analyst is first and foremost to develop vulgar competence in the field of 'study' " (emphasis in original) [3]. Developing this vulgar competence in programming poses interesting problems. Programming itself is a rather singular, concentrated craft-like activity. Although programming work is frequently done as a part of a collaborative practice, the essence of the activity still retains strong private components [5]. Extensive private 'practice' is required away from the public collaborative sphere, in a way that is perhaps rather comparable to musical instrument practice and preparation (see Sudnow's account of piano skill as a reference point [8]).

The main issue to raise in differentiating studies of programming is then the intense and unusually socially 'separate' skill involved in any collaborative work's production. Whilst ethnomethodologically-informed ethnographic workplace studies as a rule present a strong appreciation for the importance of the analyst(s) fulfilling the unique adequacy requirement, for studies of programming the problem is perhaps more acute. The myriad different programming languages on the market, for example, pose substantial problems for the analyst. Although the learning of a language typically cultivates 'transferable skills' and establishes 'common concepts,' programming languages each have their own (often arcane) specificities, and perhaps may even involve an entirely different programming paradigm altogether (e.g., object oriented versus functional programming). As such the environment available to the analyst may well be acutely less analytically habitable than a typical workplace environment.

Analytic roles and collaboration

The context of roles in which vulgar competence is required is also of importance. Broadly speaking there are two roles the analyst may assume: the role of participant in the production of code; or the role of observer to that code production.

The brief investigation and analysis presented here featured the author of this paper as both programmer and as analyst, i.e., as a 'participant' in the lived work. This dual role was performed somewhat concurrently (in the sense that 'sittings' were established) and mostly post-hoc. The role as programmer and analyst was a strongly private, rather than public and/or collaborative affair. As a result, the comprehensibility of the code document and its various sittings were highly legible, given that the author wrote the code himself. Furthermore, the code was not written in an isolation of accountability, but instead the writing of the code was made accountable to the audience of the study (rather than the audience of co-programmers on a particular project, for example).

In thinking about the analyst as participant, we must also consider a more common scenario that situates the analyst as a participant member of a team of programmers. In this case, understanding the work of co-programmers becomes a more complex and quite non-trivial issue when compared to the example of the DLA program study. For any relatively collaborative programming project, comprehension of certain parts of co-programmers' code is par for the course, however the participating analyst also has the objective of understanding programmer's *lived work* from the code document they produce. As a result much time may be spent getting a grasp on another's code from that member's perspective.

Analysts that 'observe' development in a more traditional fieldwork sense potentially encounter even greater difficulties in understanding the work of the programmers they are examining. The code document, for all its embedded legible features (whitespace, comments, structure, etc.), can often be inscrutable to those that are not direct participants either writing the very code themselves or collaborating in its use in some fashion.

Another and perhaps self-evident point to be made here is that the analysis of field data collected by the analyst might (depending upon the focus of the study) take on a *radically low-level* descriptive turn of character when compared to existing ethnographic analyses of technology, where studying records such as log files and abstract system states in order to make sense of social interaction via or around the interface is commonplace. The centrality of understanding code in understanding its production will inevitably permeate through any analytic reports, thus somewhat affecting the potential audience due to the additional expertise that may be required in reading (i.e., reading code). Sudnow's account of piano skill [8] is perhaps a analogous example of the radically low-level description that may be necessitated by any adequate excavation of the work of programmers.

Finally, the context of the program's production is also essential. The code that was inspected for analysis within this paper was logically separate and written from scratch. It was not part of a larger codebase, was largely not a piece of code that was 'glued' together from other programs, it was not produced from a detailed specification, it was not a patch for an existing codebase,

it was not produced with deployment in a real-world situation in mind, and neither was it a prototype. The way in which structure is produced in and as the daily work of the programmer(s) as well as the analysis of that code is contingent upon situation details such as these.

Capturing the lived work of programming

The code document alone is usually 'not enough' for the analyst's work, since it is a transient work-in-progress that continually changes as development progresses. In appreciating this issue, the relationship between the concrete lived work of the programmer and the code document can be considered in terms of the notion of the *lebenswelt pair* (as suggested in [1]). This expresses the idea that the code document is one half of the pair, and "makes available (in principle) for actual inspection by the reader, the real world structures of practical action" that form the programmer's day-to-day, ad-hoc working practices [4]. Even though programmers often are at pains to maintain the accountable features of their code (e.g., its 'readability'), which to some extent works in favour of the analyst and their comprehension of the programmer's work, the code document lacks substantially in the kind of detail of lived work it is the analyst's job to study in the first place. This is a common problem faced in all fields of analysis, namely the analyst's comprehension of 'what is really going on' in the work practice, meaning in this case that understanding 'what is really going on' in software development must also address the work surrounding the code document itself. So, in order for the analyst to understand what is happening at any moment in the unfolding process of developing code, these unfolding changes need to be recorded in some way. This was borne out during the problems encountered when analysing the DLA program presented here in this study.

Due to practical issues, the data collection performed during the work of writing the DLA program was limited, and did not consist of any screen-based or otherwise real-time capture of the work. Neither did it extensively include debug data and other 'side effects' of the work of programming, such as log-files. In practical terms, the major problem for capturing real-time programming work (whilst actually programming) was the highly disruptive nature of that capture during what is a concentrated activity for the programmer.

Of particular relevance to enhancing the legibility of each sitting would have been the capture of the 'edit-compile-run' cycle that is continually engaged in and commonly understood as part of 'normal' development procedure[8]. Programmers might for example edit a method or function, run the compiler on the program, run the program and inspect the output, and then return to edit informed by this output. The cyclic process ongoingly informs the development of the code across and within sittings and would have be useful for enriching an understanding of the changes between sittings.

The limited capture of sittings in this study also raise the problem of the granularity of capture as it relates to the edit-compile-run cycle of work. The fundamental problem for capture granularity is the scale of software under the analytic lense. Versioning systems such as CVS can provide very high-level

[8]Indeed, many Integrated Development Environments explicitly recognise and support this as part of the basic functionality they provide to the programmer.

records of the way in which code has developed, however clearly do not provide any detailed access. In this analysis, 'sittings' emerged as a relatively 'natural' way to break up the development cycle at a lower level than just 'versions.' However, sittings as a collection strategy may be quite artificial for other studies, especially since they can impose order externally on whatever the natural work cycle of others may be. On the other hand, given the extended and time-intensive production of code, 'low granularity' techniques such as screen-based capture only permit the collection of a relatively small window of work in terms of scope, whereas some sittings-like technique of capture can span a larger time-frame. However, even with the examination of sittings, only a relatively small segment of what was a small program (a mere 838 lines) was actually addressed as part of the analysis. Large-scale software projects may consume many 'man hours.' Issues such as these become especially pertinent in a world where massive codebases can be easily beyond a single analyst's competent understanding and familiarity. As such, a 'multi-granular' approach to capture may permit the analyst to inspect code structures in-the-large as well as the small.

Some of these various different kinds of capture are illustrated in Figure 4.

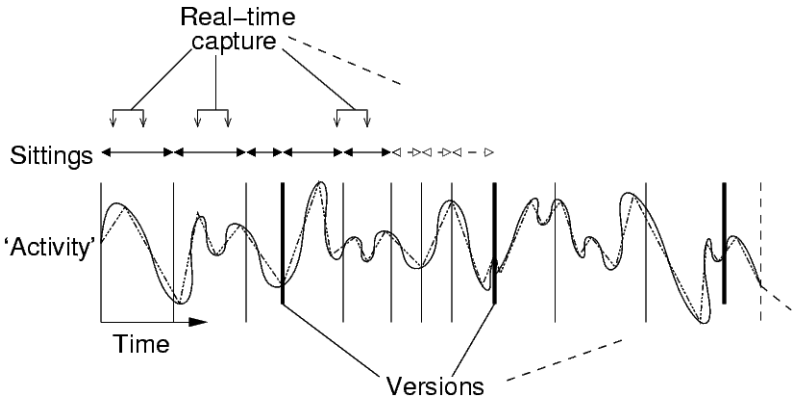

Figure 4: Different points of capture

The challenges that face the analyst have one further interesting facet to them. Typically within more accountable environments (such as a workplace) programmers heavily self-document their own work practices through the maintenance of CVS log histories, `Changelogs`, inline documentation (e.g., using `Javadoc`) and so on. It has been suggested that this documentation alone is not enough to reveal the lived practices of work, and it has also been noted that comprehension of 'what is really going on' can be contingent upon the right resources being made available. For the programmer attempting to understand a piece of code, comprehension is central as well. It would seem, then, that there is a sense in which the job of capture for analysis—i.e., making lived work available for inspection—is a similar problem to the job facing a programmer examining others' code work. The programmer all too often needs to obtain an insight into the lived work of a co-programmer's code as part of understanding that code. Thus there is some alignment between the development of programmers' tools and the analyst's tools in studying the work of programmers.

Summary

It is obvious that writing code involves more than 'just adding and removing lines of text.' Various practices are employed by the programmer in order to structure their code. Of particular interest in this paper has been the prospective nature of the way in which code is produced ongoingly, and has particularly focussed on the trajectory of the program's code as part of a series of sittings. The paper has also examined some ways in which language constructs and the programmer's practical experience of those constructs in a variety of settings may build up the programmer's working stock of knowledge about the language in which they are writing.

The difficulties present in adequately capturing this have also been discussed. One of the most important and challenging tasks presented to the analyst is developing a vulgar competence, which in this case means cultivating a deep understanding of programming from the community of programmers' perspective. Adequately capturing the lived work of the programmer has also been raised as a thorny issue given the quite 'intense' and time-consuming work involved in programming.

References

[1] B. Brown. Remarks on the foundations of computer science. In draft, available at http://www.dcs.gla.ac.uk/∼barry/papers/foundations.pdf (verified 24/03/06), 2005.

[2] G. Button and W. Sharrock. The mundane work of writing and reading computer programs. In P. ten Have and G. Psathas, editors, *Situated Order: Studies in the Social Organization of Talk and Embodied Activities*. University Press of America, Boston, 1995.

[3] A. Crabtree. Methodological issues concerning the practical availability of work-practice to EM & CA. In *Proceedings of 2nd Workplace Studies Conference, in conjunction with the Language Group of the British Sociological Association*, October 2000.

[4] A. Crabtree. Doing workplace studies: Praxiological accounts — lebenswelt pairs. *TeamEthno Online*, 1, 2001.

[5] C. R. B. de Souza, D. Redmiles, and P. Dourish. "Breaking the code," moving between private and public work in collaborative software development. In *GROUP '03: Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pages 105–114, New York, NY, USA, 2003. ACM Press.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley, 1995.

[7] L. A. Suchman. *Plans and situated actions: The problem of human-machine communication*. Cambridge University Press, 1987.

[8] D. Sudnow. *Ways of the Hand: The Organization of Improvised Conduct*. Routledge & Kegan Paul Ltd, 1978.

[9] A. Turing. Proposed electronic calculator. Technical report, Interdepartmental Technical Committee, 1945.